Understand JavaScript's "this" With Clarity, and Master It

JULY. 5 2013 190

menu

(Also learn all the scenarios when *this* is most misunderstood.)

Prerequisite: A bit of JavaScript.
Duration: about 40 minutes.

The *this* keyword in JavaScript confuses new and seasoned JavaScript developers alike. This article aims to elucidate *this* in its entirety. By the time we make it through this article, *this* will be one part of JavaScript we never have to worry about again. We will understand how to use *this* correctly in every scenario, including the ticklish situations where it usually proves most elusive.

By the founder of **JavaScriptIsSexy**

We use *this* similar to the way we use pronouns in natural languages like English and French. We write, "John is running fast because *he* is trying to catch the train."

Note the use of the pronoun "he." We could have written this: "John is running fast because **John** is trying to catch the train." We don't reuse "John" in this manner, for if we do, our family, friends, and colleagues would abandon us. Yes, they would. Well, maybe not your family, but those of us with fair-weather friends and colleagues. In a similar graceful manner, in JavaScript, we use the *this* keyword as a shortcut, a referent; it refers to an object; that is, the subject in context, or the subject of the executing code. Consider this example:

Table of Contents

```
    var person = {
    firstName: "Penelope",
    lastName: "Barrymore",
    fullName: function () {
       // Notice we use "this" just as we used "he" in the example sentence earlier?:
       console.log(this.firstName + " " + this.lastName);
    // We could have also written this:
       console.log(person.firstName + " " + person.lastName);
    }
}
```

If we use person.firstName and person.lastName, as in the last example, our code becomes ambiguous. Consider that there could be another global variable (that we might or might not be aware of) with the name "person." Then, references to person.firstName could attempt to access the firstName property from the *person* global variable, and this could lead to difficult-to-debug errors. So we use the "this" keyword not only for aesthetics (i.e., as a referent), but also fo precision; its use actually makes our code more unambiguous, just as the pronoun "he" made our sentence more clear. It tells us that we are referring to the specific John at the beginning of the sentence.

Just like the pronoun "he" is used to refer to the antecedent (antecedent is the noun that a pronoun refers to), the *this* keyword is similarly used to refer to an object that the function (where *this* is used) is bound to. The *this* keyword not only refers to the object but it also contains the value of the object. Just like the pronoun, *this* can be thought of as a shortcut (or a reasonably unambiguous substitute) to refer back to the object in *context* (the "antecedent object"). We will learn more about *context* later.

JavaScript's *this* Keyword Basics

First, know that all functions in JavaScript have properties, just as objects have properties. And when a function executes, it gets the *this* property—a variable with the **value of *the object* that invokes the function where *this* is used**.

The *this* reference ALWAYS refers to (and holds the value of) an object—a singular object—and it is usually used inside a function or a method, although it can be used outside a function in the global scope. Note that when we use **strict mode**, *this* holds the value of *undefined* in global functions and in anonymous functions that are not bound to any object.

this is used inside a function (let's say function A) and it contains the value of the **object** that invokes function A. We need *this* to access methods and properties of the object that invokes function A, especially since we don't always know the name of the invoking object, and sometimes there is no name to use to refer to the invoking object. Indeed, *this* is really just a shortcut reference for the "antecedent object"—the invoking object.

Ruminate on this basic example illustrating the use of *this* in JavaScript:

```
var person = {
firstName   :"Penelope",
lastName    :"Barrymore",
// Since the "this" keyword is used inside the showFullName method below, and the showFullName r
// "this" will have the value of the person object because the person object will invoke showl          ıe
```

```
    showFullName:function () {
    console.log (this.firstName + " " + this.lastName);
    }

    }

    person.showFullName (); // Penelope Barrymore
```

And consider this basic jQuery example with of *this*:

```
    // A very common piece of jQuery code

    $ ("button").click (function (event) {
    // $(this) will have the value of the button ($("button")) object
// because the button object invokes the click () method
    console.log ($ (this).prop ("name"));
    });
```

I shall expound on the preceding jQuery example: The use of *$(this)*, which is jQuery's syntax for the *this* keyword in JavaScript, is used inside an anonymous function, and the anonymous function is executed in the button's click () method. The reason $(this) is bound to the button object is because the jQuery library **binds** $(this) to the object that invokes the click method. Therefore, $(this) will have the value of the jQuery button ($("button")) object, even though $(this) is defined inside an anonymous function that cannot itself access the "this" variable on the outer function.

Note that the button is a DOM element on the HTML page, and it is also an object; in this case it is a jQuery object because we wrapped it in the jQuery *$()* function.

UPDATE: the following ("Biggest Gotcha" section) was added a couple of days after I published the article

The Biggest Gotcha with JavaScript "this" keyword

If you understand this one principle of JavaScript's *this*, you will understand the "this" keyword with clarity: *this* is not assigned a value until an object invokes the function where *this* is c .

Let's call the function where *this* is defined the "*this Function*."

Even though it appears *this* refers to the object where it is defined, it is not until an object invokes the *this Function* that *this* is actually assigned a value. And the value it is assigned is based **exclusively** on the **object** that invokes the *this Function*. *this* has the value of the invoking object in most circumstances. However, there are a few scenarios where *this* does not have the value of the invoking object. I touch on those scenarios later.

The use of *this* in the global scope

In the global scope, when the code is executing in the browser, all global variables and functions are defined on the *window* object. Therefore, when we use *this* in a global function, it refers to (and has the value of) the global *window* object (not in strict mode though, as noted earlier) that is the main container of the entire JavaScript application or web page.

Thus:

```
var firstName = "Peter",
lastName = "Ally";

function showFullName () {
// "this" inside this function will have the value of the window object
// because the showFullName () function is defined in the global scope, just like the firstName and la
console.log (this.firstName + " " + this.lastName);
}

var person = {
firstName   :"Penelope",
lastName    :"Barrymore",
showFullName:function () {
// "this" on the line below refers to the person object, because the showFullName function will be inv
console.log (this.firstName + " " + this.lastName);
}
}

showFullName (); // Peter Ally
```

```

// window is the object that all global variables and functions are defined on, hence:
window.showFullName (); // Peter Ally

// "this" inside the showFullName () method that is defined inside the person object still refers to the
person.showFullName (); // Penelope Barrymore
```

When *this* is most misunderstood and becomes tricky

The *this* keyword is most misunderstood when we borrow a method that uses *this*, when we assign a method that uses *this* to a variable, when a function that uses *this* is passed as a callback function, and when *this* is used inside a closure—an inner function. We will look at each scenario and the solutions for maintaining the proper value of *this* in each example.

A bit about "Context" before we continue
The *context* in JavaScript is similar to the subject of a sentence in English: "John is the winner who returned the money." The subject of the sentence is John, and we can say the *context* of the sentence is John because the focus of the sentence is on him at this particular time in the sentence. Even the "who" pronoun is referring to John, the antecedent. And just like we can use a semicolon to switch the subject of the sentence, we can have an object that is current context and switch the context to another object by invoking the function with another object.

Similarly, in JavaScript code:

```
var person = {
  firstName   :"Penelope",
  lastName    :"Barrymore",
  showFullName:function () {
// The "context"
console.log (this.firstName + " " + this.lastName);
 }
}

```

```
// The "context", when invoking showFullName, is the person object, when we invoke the shov
// And the use of "this" inside the showFullName() method has the value of the person object,
person.showFullName (); // Penelope Barrymore

// If we invoke showFullName with a different object:
var anotherPerson = {
firstName   :"Rohit",
lastName    :"Khan"
};

// We can use the apply method to set the "this" value explicitly—more on the apply () metho
// "this" gets the value of whichever object invokes the "this" Function, hence:
person.showFullName.apply (anotherPerson); // Rohit Khan

// So the context is now anotherPerson because anotherPerson invoked the person.showFullN

```

The takeaway is that the object that invokes the *this Function* is in context, and we can change the context by invoking the *this Function* with another object; then this new object is in *context*.

Here are scenarios when the *this* keyword becomes tricky. The examples include solutions to fix errors with *this*:

1. Fix *this* when used in a method passed as a callback

Things get a touch hairy when we pass a method (that uses *this*) as a parameter to be used as a callback function. For example:

```
// We have a simple object with a clickHandler method that we want to use when a button on th
var user = {
data:[
{name:"T. Woods", age:37},
{name:"P. Mickelson", age:43}
],
clickHandler:function (event) {
var randomNum = ((Math.random () * 2 | 0) + 1) - 1; // random number between 0 and 1

// This line is printing a random person's name and age from the data array
console.log (this.data[randomNum].name + " " + this.data[randomNum].age);
}
}

// The button is wrapped inside a jQuery $ wrapper, so it is now a jQuery object
// And the output will be undefined because there is no data property on the button object
$ ("button").click (user.clickHandler); // Cannot read property '0' of undefined
```

In the code above, since the button ($("button")) is an object on its own, and we are passing the *user.clickHandler* method to its click() method as a callback, we know that *this* inside our *user.clickHandler* method will no longer refer to the *user* object. *this* will now refer to the object where the user.clickHandler method is executed because *this* is defined inside the user.clickHandler method. And the object that is invoking user.clickHandler is the button object—user.clickHandler will be executed inside the button object's click method.

Note that even though we are calling the clickHandler () method with **user**.clickHandler (which we have to do, since clickHandler is a method defined on user), the clickHandler () method itself will be executed with the button object as the context to which "this" now refers. So *this* now refers to is the button ($("button")) object.

At this point, it should be apparent that when the context changes—when we execute a method on some other object than where the object was originally defined, the *this* keyword no longer refers to the original object where "this" was originally defined, but it now refers to the object that invokes the **method** where *this* was defined.

Solution to fix *this* when a method is passed as a callback function:
Since we really want this.data to refer to the data property on the *user* object, we can use the Bind (), Apply (), or Call () method to specifically set the value of *this*.

I have written an exhaustive article, JavaScript's Apply, Call, and Bind Methods are Essential for JavaScript Professionals, on these methods, including how to use them to set the *this* value in various misunderstood scenarios. Rather than re-post all the details here, I recommend you read that entire article, which I consider a must read for JavaScript Professionals.

To fix this problem in the preceding example, we can use the bind method thus:

Instead of this line:

```
$ ("button").click (user.clickHandler);
```

We have to **bind** the clickHandler method to the user object like this:

```
$("button").click (user.clickHandler.bind (user)); // P. Mickelson 43
```

— View a working example of this on JSBin

2. Fix *this* inside closure

Another instance when *this* is misunderstood is when we use an inner method (a closure). It is important to take note that closures cannot access the outer function's *this* variable by using the *this* keyword because the *this* variable is accessible only by the function itself, not by inner functions. For example:

```
var user = {
tournament:"The Masters",
data      :[
{name:"T. Woods", age:37},
{name:"P. Mickelson", age:43}
],

clickHandler:function () {
// the use of this.data here is fine, because "this" refers to the user object, and data is a property

this.data.forEach (function (person) {
// But here inside the anonymous function (that we pass to the forEach method), "this" no longe
// This inner function cannot access the outer function's "this"

console.log ("What is This referring to? " + this); //[object Window]

console.log (person.name + " is playing at " + this.tournament);
// T. Woods is playing at undefined
// P. Mickelson is playing at undefined
})
}

}

user.clickHandler(); // What is "this" referring to? [object Window]
```

this inside the anonymous function cannot access the outer function's *this*, so it is bound to the global window object, when *strict* mode is not being used.

Solution to maintain *this* inside anonymous functions:
To fix the problem with using *this* inside the anonymous function passed to the *forEach* method, we use a common practice in JavaScript and set the *this* value to another variable before we enter the forEach
method:

```
var user = {
tournament:"The Masters",
data      :[
{name:"T. Woods", age:37},
{name:"P. Mickelson", age:43}
],

clickHandler:function (event) {
// To capture the value of "this" when it refers to the user object, we have to set it to another var
// We set the value of "this" to theUserObj variable, so we can use it later
var theUserObj = this;
this.data.forEach (function (person) {
// Instead of using this.tournament, we now use theUserObj.tournament
console.log (person.name + " is playing at " + theUserObj.tournament);
})
}

}

user.clickHandler();
// T. Woods is playing at The Masters
//  P. Mickelson is playing at The Masters
```

It is worth noting that many JavaScript developers like to name a variable "that," as seen below, to set the value of *this*. The use of the word "that" is very awkward for me, so I try to name the variable a noun that describes which object "this" is referring to, hence my use of *var theUserObj = this* in the preceding code.

```
// A common practice amongst JavaScript users is to use this code
var that = this;
```

— View a working example of this on JSBin

3. Fix *this* when method is assigned to a variable

The *this* value escapes our imagination and is bound to another object, if we assign a method that uses *this* to a variable. Let's see how:

```
// This data variable is a global variable
var data = [
{name:"Samantha", age:12},
{name:"Alexis", age:14}
];

var user = {
// this data variable is a property on the user object
data    :[
{name:"T. Woods", age:37},
{name:"P. Mickelson", age:43}
],
showData:function (event) {
var randomNum = ((Math.random () * 2 | 0) + 1) - 1; // random number between 0 and 1

// This line is adding a random person from the data array to the text field
console.log (this.data[randomNum].name + " " + this.data[randomNum].age);
}

}

// Assign the user.showData to a variable
var showUserData = user.showData;

// When we execute the showUserData function, the values printed to the console are from the g
//
showUserData (); // Samantha 12 (from the global data array)

```

Solution for maintaining *this* when method is assigned to a variable:
We can fix this problem by specifically setting the *this* value with the bind method:

```
// Bind the showData method to the user object
var showUserData = user.showData.bind (user);

// Now we get the value from the user object, because the <em>this</em> keyword is bound to
showUserData (); // P. Mickelson 43
```

4. Fix *this* when borrowing methods

Borrowing methods is a common practice in JavaScript development, and as JavaScript developers, we will certainly encounter this practice time and again. And from time to time, we will engage in this time-saving practice as well. For more on borrowing methods, read my in-depth article, JavaScript's Apply, Call, and Bind Methods are Essential for JavaScript Professionals.

Let's examine the relevance of *this* in the context of borrowing methods:

```
// We have two objects. One of them has a method called avg () that the other doesn't h
// So we will borrow the (avg()) method
```

```
var gameController = {
scores  :[20, 34, 55, 46, 77],
avgScore:null,
players :[
{name:"Tommy", playerID:987, age:23},
{name:"Pau", playerID:87, age:33}
]
}

var appController = {
scores  :[900, 845, 809, 950],
avgScore:null,
avg     :function () {

var sumOfScores = this.scores.reduce (function (prev, cur, index, array) {
return prev + cur;
});

this.avgScore = sumOfScores / this.scores.length;
}
}

//If we run the code below,
// the gameController.avgScore property will be set to the average score from the appController

// Don't run this code, for it is just for illustration; we want the appController.avgScore to remain
gameController.avgScore = appController.avg();

```

The avg method's "this" keyword will not refer to the gameController object, it will refer to the appController object because it is being invoked on the appController.

Solution for fixing *this* when borrowing methods:
To fix the issue and make sure that *this* inside the appController.avg () method refers to gameController, we can use the *apply ()* method thus:

```

// Note that we are using the apply () method, so the 2nd argument has to be an array—
```

```
appController.avg.apply (gameController, gameController.scores);

// The avgScore property was successfully set on the gameController object, even though we bor
console.log (gameController.avgScore); // 46.4

// appController.avgScore is still null; it was not updated, only gameController.avgScore was upda
console.log (appController.avgScore); // null
```

The *gameController* object borrows the appController's avg () method. The "this" value inside the appController.avg () method will be set to the gameController object because we pass the gameController object as the first parameter to the apply () method. The first parameter in the apply method always sets the value of "this" explicitly.

— View a working example of this on JSBin

Final Words

I am hopeful you have learned enough to help you understand the *this* keyword in JavaScript. Now you have the tools (bind, apply, and call, and setting *this* to a variable) necessary to conquer JavaScript's *this* in every scenario.

As you have learned, *this* gets a bit troublesome in situations where the original context (where *this* was defined) changes, particularly in callback functions, when invoked with a different object, or when borrowing methods. Always remember that *this* is assigned the value of the object that invoked the *this Function*.

Be good. Sleep well. And enjoy coding.